CSE 70: Design Patterns & Software Architecture
Adapter, Observer, MVC, Layers, Client/Server,
Pipes & Filters, Tiers

Ingolf Krueger

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0114, USA

California Institute for Telecommunications
and Information Technologies
La Jolla, CA 92093-0405, USA
Learning Goals for Today
Learning Goals

• Be able to identify basic patterns in given design problems

• Know the term “Software Architecture” and its influence on software quality

• Understand the structure, behavior and use of the following Architecture Patterns and Styles:
  - MVC
  - Layered Architecture
  - Client/Server
  - Pipes & Filters

• Know the “4+1” model of software architecture

• Know the term “Refactoring” and what it means
Adapter Pattern
Example

Make the Client and Server classes work together.

Assume that Message has a toString() method.
Team up for 7 minutes!

```java
public class Server {
    ...
    public Message[] getMessages() {
        ...
    }
    ...
}

public class Client {
    ...
    private String[] messages;
    ...
    public readAll() {
        messages = server.getMessages();
        ...
    }
}
```
• **Context & Problem**
  - A given component (*target*) offers the desired functionality
  - The target's client expects a different interface

• **Solution:**
  - Define an adapter component that offers the required interface
  - Associate the client with the adapter instead of with the target
  - Let the adapter relay calls from the client to the target; adjust the relayed calls towards the interface of the target
  - Let the adapter provide functions missing in the target
Adapter: Structure

Client

TargetComponent

DesiredInterface

Adapter

service()

service()

TargetComponent

targetService()

service() {return target.targetService();}
Adapter: Behavior
Observer Pattern
Chat System Observer Pattern: Behavior

```
c2:Client
     dispatchMessage (send_msg)
         send()
         notify(notification)
     notify(notification)

:Server

     send()

     notify(notification)

     send()

     notify(notification)
```

© Ingolf Krueger, 2010
Observer

- **Context & Problem**
  - Several components depend on a target component’s state
  - All dependent components shall be informed about state changes of the target
  - Loose coupling between dependent and target component target required

- **Solution:**
  - Let all dependent components attach/register with the source
  - Let components no longer interested in the target’s state changes detach/unregister from the target
  - In the event of a state change let the target send a notification message to all registered components
  - Let the registered components inquire about the new state after they have been notified
public class BaseSource{
    IObserver[] observers;
    public void attach(IObserver) {...}
    public void detach(IObserver) {...}
    public void update() {...}
}

public interface IObserver {
    public void notify();
}
Observer: Structure

- **BaseSource**:
  - `attach(IObserver)`
  - `detach(IObserver)`
  - `update()`

- **Source**:
  - `state`
  - `getState()`
  - `setState()`

- **Observer**:
  - `notify()`

- **IObserver**:
  - `notify()`

- **Interface**:
  - `for all o in observers {
      o.notify();
    }

- **Client/Observer**

- **Source/Target**
public class Source extends BaseSource {
    State state;
    public State getState() {...}
    public void setState(State s) {...}
    ... }

public class Observer implements IObserver {
    public void notify() {
        ... }
    }

Observer: Structure

- BaseSource
- State
- getState
- setState
- public class Source extends BaseSource
- Observer
- public class Observer implements IObserver
- notify

© Ingolf Krueger, 2010
Observer: Behavior

- `:Source`
- `o1:Observer`
- `o2:Observer`

Methods:
- `setState(s)`
- `update()`
- `notify()`
- `getState()`

Diagram shows the interaction between the `Source` and `Observers`.
• Consequences:
  - Loose coupling between target and dependent components
  - Support for event-based systems (multithreading)
  - Design strategy for multicast-systems
  - Problematic: Containment of update frequency

Solution: apply multiple update strategies
(Strategy pattern)
Your turn!
Try to find the pattern (structure/behavior) in the following two scenarios
Team up with your neighbor for 10 minutes

**Title:** Browse Folders on Desktop  
**Description:** User opens two separate windows W1 and W2, both showing folder F. In W1, user creates a new file. The file appears automatically also in W2.

**Title:** Viewing Election Results  
**Description:** On election night, results come in and are stored in a spreadsheet. One TV moderator shows them using a “magic wall” in bar chart form. Another TV moderator clicks on his laptop to show a pie chart with the identical results.
Model, View, Controller (MVC)
MVC

• Context & Problem
  - Application data needs to be maintained and presented via the user interface
  - Multiple different output formats need to be supported
  - Multiple different forms of input need to be supported
  - How to establish consistency?

• Solution:
  - Introduce separate components for storing and processing of data (model), data presentation (view), and for handling input (controller)
  - The model represents the functional core; it registers dependent components (views and controllers) and notifies them about data changes
  - The view retrieves data from the model and displays it
  - The controller translates user input into events for the model; it may also change the UI to mirror data changes
MVC: Example

Model: user/buddy database, chat logs, …
MVC: Roles and Responsibilities

- **User:**
  - Gives input via controller
    (by typing, speaking, gesturing etc. – input modalities)

- **Model:**
  - holds the state of the application
  - notifies controllers/views when state changes

- **Controller:**
  - Receives input from user and translates it into internal commands to change the state of the model
  - Different controllers can respond to different input modalities (one for typing, one for speaking, etc.)
  - Sets the “next” appropriate view and controller based on command and current view/controller

- **View:**
  - displays state to the user
  - Different views can exist at the same time
MVC: Example (Reality: multiple Controllers)

Model: user/buddy database, chat logs, …
MVC: Structure

BaseSource
- attach(Observer)
- detach(Observer)
- update()

Observers
- notify()

Interface

Model
- state
- getState()
- setState()
- service()

View
- display()

Controller
- handleInput()
MVC: Behavior

- Controller: `handleInput()`, `notify()`, `getState()`
- Model: `service()`, `update()`, `notify()`, `getState()`
- View: `display()`
MVC

• Consequences:
  - Decoupling of application data from presentation and input mechanism
  - Consistency of user interface and underlying data model
  - Increase of structural and dynamic complexity
  - Potential loss of performance
Software Architecture
Architecture? What Architecture?

GUI

GUI-Coupling

Application Server

Middleware

Hardware Abstraction

Hardware

Legacy

© Ingolf Krueger, 2010
Architecture Improvement

© Ingolf Krueger, 2010
The Notion of Architecture

A Software-Architecture describes the decomposition of a system into

*units / components / subsystems,*

and their

*connections / interactions / relationships*

observing

*quality requirements / design guidelines / constraints*

More precisely: interfaces (offered/used)

- Syntax (signals/methods, typing)
- Behavior (protocols)

Often:

(arbitrary) distinction between “functional” und “non-functional” properties
Architecture Patterns
What are Architecture/Design Patterns?

- Describes one proven solution for a recurrent design problem
- Defines the context for the solution’s applicability

Adapted from [POSA96]
"An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

[POSA96]
Example: MVC

Model: user/buddy database, chat logs, ...

© Ingolf Krueger, 2010
Example: Client/Server

```
Computer 3
  JVM 3
    server: ChatServer

Computer 1
  JVM 1
    client1: ChatClient

Computer 2
  JVM 2
    client2: ChatClient
```
Example: Layered Architecture

- Application
- Presentation
- Business logic
- Database
- Link
- Physical

Layer n

Layer n+1

Generalization yields pattern “layered architecture”
Example: “Pipes & Filters”
Example: Compiler

Source Code → Scanner → Parser → Code-Generator → Target Code

Intermediate Steps:
- Token
- Abstract Syntax Tree
- Symbol-Table
- Error Trace
- Error Output

Error Messages
Example: “Interpreter”

```
[Diagram]
```

- **Program**
- **Virtual Machine**
- **Base System**

“User”
Variation on Client/Server: DB-Centric Design
Architecture & Quality
The Role of SW-Architecture

• The selection of an adequate architecture is a key success factor in system design

• Transparantly structured architecture is basis for:
  – System understanding
  – Project organization
  – Complexity management
  – Reuse
  – Manageable system evolution

⇒ Make architecture description part of every software project!
Architecture Influences System Properties

- reusable
- changeable
- portable
- testable
- implementable
Architecture Influences System Properties

- functional
- extensible
- user friendly
- performant
- economical
- available
- safe/secure
- dependable
- scalable
Architectural Aspects
Architectural Aspects: 4+1 model*

* adapted from Philippe Kruchten: The Rational Unified Process – An Introduction, Addison-Wesley, 1999

- Functional requirements
  - class/sequence/activity/
    - ... diagrams

- Source code organization
  - File structure
  - Configuration information
  - ... user stories
  - concurrency
  - Response times
  - Throughput
  - ... process view

- Mapping of executables to processors
  - System platform
  - Installation
  - ... deployment view

© Ingolf Krueger, 2010
Example: OOI
Example from the “real world”: Ocean Observatories Initiative
“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.”

*M. Fowler et al.: Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999*
Refactoring:  
- “minimally invasive” modifications to system structure  
  - Strategy of small steps  
- Set up adequate test suite before changing the system  
- Carry out tests during and after performing the change  
  - Increases confidence in correctness;  
    Goal: no change of observable behavior
Refactoring Techniques

Example: Move method to superclass

```
MessagePrinter
  ↓
StandardPrinter
  print()
PrettyPrinter
  print()
```

"Pull Up Method"

```
MessagePrinter
  ↓
StandardPrinter
PrettyPrinter
```

© Ingolf Krueger, 2010
Refactoring Techniques

• Refactoring proceeds in extremely small steps

• Each individual change is manageable

• Refactoring fits particularly well with design patterns and architectural patterns:

  1. Evaluate current state

  2. Select target pattern

  3. Identify sequence of refactoring steps leading to target pattern

  4. Perform refactoring until target pattern is reached
What have you learned today?
Learning Goals

• Be able to identify basic patterns in given design problems

• Know the term “Software Architecture” and its influence on software quality

• Understand the structure, behavior and use of the following Architecture Patterns and Styles:
  – MVC
  – Layered Architecture
  – Client/Server
  – Pipes & Filters

• Know the “4+1” model of software architecture

• Know the term “Refactoring” and what it means